(2)

An Experiment in Table Driven Code Generation

Technical Report

R.S. Fabry
(415) 642-2714

Contract N00039-82-C-0235
November 15, 1981 - September 30, 1983

ARPA Order Number 4031

Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.

An Experiment in Table Driven Code Generation[1]

by

Susan L. Graham
Robert R. Henry
Robert A. Schulman

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

Abstract

We have constructed a local code generator for the VAX-11[2] using a parser-like instruction pattern matcher. The code generator replaces the second pass of the UNIX[3] Portable "C" compiler. This paper describes the design of the code generator and the special considerations imposed by the pattern matching process. We summarize the structure of the machine description grammar and its associated semantic actions, as well as the tools we developed to manipulate the large VAX description. In our experience, this approach makes the instruction selection phase of the compiler easier and faster to implement, and more likely to be correct than traditional techniques.

1. Introduction

In the last few years, we have been studying an approach to code generation in which instructions are selected by a pattern-matching process that chooses instructions from a table generated automatically from a machine description. The initial version of the technique is described in [Glanville77] and [Glanville78]. [Graham80] gives an overview of this and related techniques; [Henry81] gives a detailed description of an implementation, together with some improvements to the method. Related work appears in [Ganapathi80], [Jansohn80], [Gujral81], and [Crawford82].

Recently, we conducted a limited experiment to gain more experience with this technique [Schulman82]. Our goal was to attempt to use this method in a controlled setting in which comparisons would be possible. Our experience has been favorable and is described in this paper.

[1]Support for this work was provided by the National Science Foundation under grant MCS80-05144.

[2]VAX is a trademark of Digital Equipment Corporation.

[3]UNIX is a trademark of Bell Laboratories.

2. The Experiment

The UNIX portable "C" compiler (abbreviated as PCC) [Johnson79] is a retargetable compiler for the systems programming language "C". It is written in "C" and runs under the UNIX operating system. PCC has been successfully retargeted to over two dozen machines [Johnson81]. The compiler has two logical phases. The first phase is mostly specific to "C", and produces an intermediate representation of the program consisting of a forest of expression trees interspersed with target machine specific instructions to implement unconditional control flow. The second phase generates assembly code and is mostly target machine dependent, driven by a somewhat *ad hoc* pattern matcher using patterns taken from a hand generated table. PCC does no global optimization or recognition of common subexpressions. A Fortran 77 compiler [Feldman79] and the Berkeley Pascal compiler [Joy79] share the same intermediate representation and second pass of the portable "C" compiler.

Our experiment w. to replace the second pass of the portable "C" compiler with a code generator that uses the Graham-Glanville methods. We chose to produce code for the machine we had available, a VAX-11, and to devote only a limited amount of time to the project. Our initial goal was not to develop a truly retargetable compiler, but to gain experience with Graham-Glanville techniques, generating good local code for a real machine, for production programming languages, to force us to solve real problems. By comparing our generated code with that produced by PCC, we could analyze the effectiveness of our implementation decisions. This goal has been achieved, and with this experience we are now trying to make the code generator retargetable and to improve its performance.

We had available a previous implementation of a Graham-Glanville code generator generator, the Code Generator Generator's Work Station (CGGWS) [Henry81]. However, we chose to discard parts of it. The CGGWS's internal model of instruction descriptions was too limited; it ran too slowly and produced tables that were too large. The CGGWS's code generator spent too much time interpreting cumbersome tables when checking semantic attributes. When we began the experiment we were uncertain how we wanted to change the automated semantic handling. Consequently, we chose to program the semantic attribute evaluator as VAX-specific routines hand-coded in "C", rather than attempting to generate the semantic actions automatically. We are now converting to program-generated semantic actions. This change will make retargeting easier.

We were able to reclaim all parts of the CGGWS relating to the *syntactic* aspects of a machine description grammar, including the table generator. All pieces of the CGGWS dealing with *semantic* aspects were discarded.

3. Graham-Glanville Code Generation

There are three logical parts to the basic Graham-Glanville code generation technique. The target machine description (1) is fed to a table constructor (2) in the code generator generator. The constructed tables then control an instruction pattern matcher (3) in the code generator. The first two parts are *static*; they are used once for each target machine. The last part is *dynamic*, invoked for every program being compiled.

3.1. Target Machine Description

In the Graham-Glanville system, instructions on the target machine are described as attributed productions in a context free grammar. There is one non-terminal in the grammar for each class of registers on the target machine, and an additional sentential non-terminal. In addition, there can be non-terminals for pseudo-registers and for status bits or condition codes. The terminal symbols in the grammar are the node labels in the expression trees for which code is to be generated. The right hand sides of the productions in the grammar have the prefix linearized form of a computation tree consisting of terminals and non-terminals. The left hand side designates how the computation effects the processor. In the original Graham-Glanville formulation, each production describes the operation performed on the target machine by *one* instruction, together with *one* combination of addressing modes describing the operands. We call such a machine description grammar *flat*.

For most examples in this paper, we will show the linearized tree, instead of the graphical representation of the tree. Figure 1 describes the terminal and non-terminal symbols we will use throughout the remainder of the paper (except for the Appendix). By convention, all terminal symbols start with an upper case letter; non-terminal symbols begin with lower case letters. Unless specified otherwise, all non-terminal symbols are nil-ary.

symbol	# (arity)	meaning	left child	right child
Assign	2	assign	destination	source
Plus	2	add	operand	operand
Mul	2	multiply	operand	operand
Cbranch	2	conditional branch	test	destination
Cmp	2	compare	operand	operand
Indir	1	memory fetch	address	
Name	0	global variable		
Dreg	0	dedicated register		
Zero	0	0		
One	0	1		
Two	0	2		
Four	0	4		
Eight	0	8		
Const	0	constant		
Label	0	label		
rval	0	source		
lval	0	destination		
reg	0	register		

Figure 1: Terminal and Non-Terminal Symbols

As described in [Glanville78], each production has associated with it a string and a set of semantic bindings into the left and right hand sides of the production. The string and bindings are used to construct the assembly code representation of the generated instructions. In addition, each symbol on the right hand side may have semantic qualifications, all of which must be met before that pattern is selected. For example, the semantic qualification may require a constant to be in a specified range (implementing a *range idiom*), or require a constant or register to be the same one referenced in another part of the production (implementing a *binding idiom*). As the reader will see, semantic qualification is handled differently in our code generator.

3.2. Table Constructor

The machine description grammar is processed by a table-generating program similar to an SLR(1) parser generator. Most machine description g[illegible] are highly ambiguous, since the target mac[illegible]sually implement an expression in many dif[illegible]s. The table generator disambiguates the grammar by favoring a shift over a reduce in a shift/reduce conflict, and a reduction by the longest possible rule in a reduce/reduce conflict. Thus, the table-driven pattern matcher implements the *maximal munch method* [Cattell80] in which the largest possible pattern is matched. If there are two or more longest rules, then the table generator cannot statically choose among them. In that case, the pattern matcher will choose among them dynamically using semantic attributes.

The table generator contains algorithms to ensure that the pattern matcher will not get into a looping configuration, where non-terminal chain rules are cyclically reduced. The table generator also checks if there is some input for which the pattern matcher will perform an error action, also called a *syntactic block*. In this case, the present table generator only notifies the user, and does not attempt corrective action.

In addition, the table generator looks for patterns which can be used only if a semantic condition is met. In general, the input cannot be guaranteed to satisfy such a condition. Therefore, in the event of such a *semantic block*, the table generator constructs a *default action* alternative from other productions. That default alternative typically causes more than one instruction to be generated.

3.3. Pattern Matcher

The instruction pattern matcher is a table-driven shift/reduce parser, invoked once for each expression to be compiled. Each reduction corresponds to one logical instruction[4] emitted in linear time in a provably correct order. Since the pattern matching and table construction algorithms are based on a well understood model, and the tables are constructed automatically by a provably correct algorithm [Glanville78], the only source for error in constructing the instruction selector is in the machine description or the supporting semantic routines.

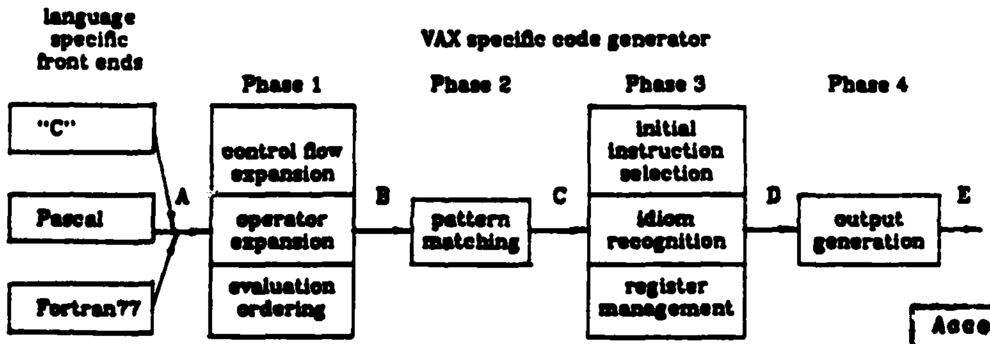
4. Modifications to the Basic Algorithm

We find that because of the richness of the instruction set, a flat machine description grammar for the useful VAX instructions, written using only the register and the sentential non-terminals, would have over 8 million productions. In order to write and process a feasible VAX description, the description grammar must be *factored*. In a factored grammar, additional non-terminals group together symbols having a common function throughout the grammar. To preserve the properties of the pattern matcher, only two kinds of factoring are considered: factoring of complete subtrees and factoring of operator symbols into classes. Thus, in a factored grammar, each right hand side is either a flattened tree or a single operator symbol.

Not every production in this factored grammar causes code to be emitted. Productions now either *encapsulate* phrases (subtrees), *emit* instructions, or serve as *glue*. In the factored grammar, a non-terminal no longer has a simple semantic attribute, as it did in the flat grammar when it could only be a register. With the freedom to factor, there are now many more choices to make when writing the grammar and implementing the semantics. We will describe the design decisions for the VAX grammar and semantic attributes we used after we outline the phase structure of the code generator.

5. Code Generator Organization and Data Flow

In order to cater to the needs of the parser-driven pattern matcher, our code generation phase is one single program structured into logical subphases, as shown in Figure 2. Roughly one half the code generation time is spent in the pattern matching phase. We now describe the phases in more detail.



Interphase objects

A	an expression tree from "C", Pascal or Fortran 77
B	sequence of expression trees
C	sequence of patterns
D	operator and operand semantic descriptors
E	symbolic VAX assembly code

Figure 2: The Code Generator's Logical Subphases

[4] This logical instruction may really be a pseudo-instruction, corresponding to several hardware instructions.

5.1. Phase 1: Tree Transformation

In the first phase, each expression tree for which code is to be generated is transformed to make code generation easier. Some of these transformations might more properly be done by the *first pass* of the "C", Pascal or Fortran compilers. However, we chose not to modify the existing "front ends".

5.1.1. Phase 1a: Explicit Control Flow

The short circuit operators in "C" (&& and ||) contain implicit control flow. Sub-trees with these operators are rewritten to make the implicit tests and conditional branches explicit. Sub-trees describing function calls are replaced by compiler generated temporaries so that context switching does not occur within expression trees. A use of such a temporary is preceded by an expression tree which assigns the result of the function evaluation to the temporary. (Thus, function calls are no longer embedded in more complex expressions.) Both of these transformations are target machine independent.

Two additional control flow transformations are motivated by the architecture of the VAX. The first removes selection operators from the tree. The selection operators test a boolean value and evaluate one of two possible expressions for the value of the selector. A transformation makes the conditional branches explicit. Second, a truth value representing the result of a comparison must be constructed by a sequence of tests, jumps and assignments, since the VAX lacks an instruction to construct such a value. The tree is rewritten to insert the more primitive operations.

These operators are rewritten in phase one because we wanted to isolate all issues of control flow to the tree rewriting phase. Unfortunately, the latter two transformed computations each need a temporary register. Therefore, the first phase requires a register manager that is totally disjoint from the register manager in the third phase. This tradeoff needs to be reevaluated; see §5.3.3.

5.1.2. Phase 1b: Operator Expansion and Canonically Ordered Operands

Some additional transformations are motivated by the particular target machine. Rewriting is necessary to expand intermediate language operators having no corresponding hardware operator on the target machine. Unsigned arithmetic operators are typical examples.

We also rewrite the tree to reorder certain groups of operands and replace particular computations by others so that fewer patterns are needed. This *canonicalization* reduces the number of patterns significantly, especially those describing addresses. For example, left shift by a constant is replaced by multiplication by the appropriate power of 2, subtraction by a constant is replaced by an addition, and a constant operand child of an addition operator is forced to be the left child. We could also perform constant folding at this point, but we assume that the "front ends" have done so.

5.1.3. Phase 1c: Evaluation Ordering

Before instructions are selected for an expression tree, the portable "C" compiler reorders the tree, using two goals motivated by the techniques in [Sethi70]. *The first goal is to minimize the register* requirements for evaluating the expression tree. The second goal is to discover subexpressions whose evaluation will cause subsequent register spilling and to insert explicit stores to compiler generated temporaries to avoid the spill. By handling the situation in advance, the code selector will never run out of registers. Since in the portable "C" compiler reordering is distinct from instruction selection, coordination is less than perfect, and is a source of many errors in retargeting PCC [Johnson81]. Nevertheless, the attempt to prevent spilling is a good idea. The expression reordering in PCC occurs in its second pass and is therefore not reflected in the input to our code generator.

Since the instruction selector in our code generator does a left to right, no backup traversal of the expression tree, unpredictably allocating registers, it is possible that a mostly right recursive tree could run out of registers. However, an equivalent left recursive tree might not have this problem. Since "C", Pascal and Fortran do not specify an operand evaluation ordering, we introduced a simple reordering heuristic in the first phase of our code generator. The heuristic is to assume that the more complicated subtree of a binary operator, and hence the one that should be the left subtree, is the subtree with the most nodes. The subtrees are swapped according to this measure, and if the binary operator is not commutative (assignment, subtraction and division are examples), then it is replaced by a new operator that tells the third phase of the code generator to order the computed values properly. In our experiment, adding these reverse binary operators increased the size of the grammar by 25%, increased the size of the tables by 60%, but affected register allocation in less than 1% of the expressions in one set of "C" programs. These statistics reflect both the simple nature of "C" expressions, and the preexisting left recursive bias of compiler-generated expression trees.

The first phase of our code generator will discover those subtrees which will always require register spilling on the VAX, by virtue of the operation being performed. However, because of register management complexity issues, the first phase can not know which values *might* be spilled. Since function calls and structure or record assignment always require spills, the first phase factors them out of expressions and replaces them with references to compiler temporaries. The register manager in the third phase is prepared to spill registers into temporaries on the fly.

5.2. Phase 2: Pattern Matcher

The second phase is the pattern matcher. Within the pattern matcher, each encapsulating reduction condenses the semantic attributes of the pattern into a *signature* associated with the left-hand side non-terminal. Typically, these reductions correspond to addressing modes. Since the other phases neither can, nor should, predict how the pattern matcher will make these condensations, all communication from the preceding tree transformers to the following semantic phase is through the semantic attributes. This convention can be a logical bottleneck. Other reductions are either for parsing purposes, or cause one or more instructions to be emitted by the subsequent phase.

5.3. Phase 3: Instruction Generation

The majority of the work in the post-pattern-matching phase performs relatively uninteresting housekeeping chores. The interesting parts are the semantic aspects of instruction selection and generation, and register allocation.

As a precondition to entry into the instruction generation phase, the syntactic pattern has selected a syntactic pattern for a three address instruction. That pattern may correspond to more than one choice of instructions or pseudo instructions, depending on the semantic restrictions. The pattern matcher can, therefore, be regarded as an instruction *schema*. If it was necessary, the pattern matcher has also forced the instruction's operands to be converted to the appropriate data types.

5.3.1. Phase 3a: Initial Instruction Selection

Instruction selection is driven by the selected syntactic *pattern*, and by the information stored in a hand written *instruction table*. Each entry in the instruction table distinguishes among different instructions having the same syntactic description, and specifies the description of each instruction that can be emitted. An entry in this table is chosen based on the generic operator and the types of its operands.

Figure 3 shows the table entry for addition of longs. The "op" field is the operator name. The "#" field is the number of operands required by the operator and the "print" field is the assembler mnemonic for the instruction. The "binding" field specifies the binding idiom (an operator name), the "↔" field indicates whether the source operands can be swapped, and the "range" field specifies the internal name for a range idiom.

op	#	type	print	binding	↔	range
ADD	3	LONG	"addl3"	ADD	yes	l_add3
ADD	2	LONG	"addl2"	null	no	l_add2
INC	1	LONG	"incl"	null	no	null

Figure 2: Instruction Table Entry for Long Addition

When requested to generate code for the (atypical) "C" assignment expression

a = 17 + a;

the instruction selector is presented with the syntactic pattern for a three address add instruction, and these semantic descriptors for the operator and the operands:

generic operator	destination operand	1st source operand	2nd source operand
ADD	<L, a(*fp*)>	<L, $17>	<L, a(*fp*)>

(Here, "a(*fp*)" is the assembler syntax for variable "a", "$17" is the assembler syntax for the immediate constant "17", and "L" stands for the data type long.)

Initially, the first line in the instruction table entry is found. The idiom recognizer may then select the second or third line in the instruction cluster. We will return to this example in the next section.

If semantic blocking were possible for a given instruction, the instruction selector would first check the semantic restrictions causing blocking. If there were a semantic block, the selector would then replace the given instruction by a default list. However, our VAX description does not contain any semantic blocks.

If the destination were an assignable register, instead of a memory location, then the register manager would be called at this point. The assigned register would be encapsulated into an addressing mode descriptor, and the general instruction selection mechanism would be called.

5.3.2. Phase 3b: Idiom Recognition

We recognize two classes of idioms, binding idioms and range idioms. A binding idiom determines whether one of the source operands matches the destination operand, as one precondition for turning a three address instruction into a two address instruction. A range idiom checks whether one of the sources to the instruction is a constant in a particular, possibly degenerate, range. Binding idioms are always checked before range idioms. There is one range idiom associated with each VAX instruction that can be simplified. The range idioms are implemented by functions written in "C"; these functions follow a relatively straightforward coding style.

Let us now return to the example. The binding idiom (ADD) checks that one of the sources matches the destination. Either source will do, as specified by the "↔" field in the instruction table. Since the second source matches the destination, the two operand variant is used. The two operand variant has no binding idiom, but there is a range idiom associated with the addl2, so the range idiom is tried. That idiom determines whether the other source is a literal with the value one; if so, the incl instruction would be used. Since in our example the test fails, the addl2 instruction is emitted instead.

Since we have the mechanisms, it is convenient to have the idiom recognizer catch certain pseudo-instructions. The previous phases incorrectly assume that the VAX can implement these pseudo-instructions in one machine instruction. These pseudo-instructions include signed integer modulus, which requires a register to hold an intermediate result, and unsigned division which requires a call to a library function that is known not to modify any registers. These operators are not rewritten using the general mechanism in the first phase, because they were not anticipated in the design of that phase.

With the exception of pseudo-instruction expansion, the idiom recognizer sub-phase is optional in the sense that if it were omitted, correct code would still be generated. Many of the choices made by this phase could instead be made by a more general peephole optimizer, provided that register assignment had not already been done.

5.3.3. Phase 3c: Register Management

The register manager is extremely simple and unsophisticated. The conventions for register usage established by the portable "C" compiler divide the registers for the VAX into *dedicated* registers, which are assigned by the first pass of PCC, and the *allocatable* registers, which are assigned by our register manager. The register manager in this phase supplies assignable registers upon demand from other routines in the instruction generator.

Since there is no detection of common sub-expressions (except those with very short lifetimes created by the tree transformer), the least recently used register is also the register with the most distant future use. Consequently, the registers can be assigned and freed with a stack discipline. When an assignable register is requested as a destination for a particular instruction, the register manager attempts to reclaim and reuse assignable registers from the source operands to the instruction. Failing that, the next free register is selected. If there is no allocatable register available, a register from the bottom of the stack is spilled. Registers are always spilled to compiler generated variables. We call memory locations holding spilled values *virtual registers*, to distinguish them from *physical registers*.

If a register is spilled, it is reloaded just before it is used, since the register manager cannot determine whether in a given context an alternative instruction could have fetched the operand directly from memory[6]. No attempt is made to remember register contents in order to replace fetches of copies from memory or to avoid spills when copies already exist in memory.

Recall that register assignment is also performed when rewriting trees in the first phase (§5.1.3). Despite logical separation, both phases allocate registers from the same hardware register bank, so the effects of register assignment performed in the first phase must be communicated to this phase. The first phase generates special trees specifying which registers it assigned, as well as a use count. The description grammar has special productions to match these trees, and the register manager in this phase adjusts its internal model accordingly.

We successfully ran and developed the code generator for months without finding a "C" or Pascal program that ran out of registers. Consequently, it seemed unimportant to implement better register management techniques. Finally, the demands of certain Fortran programs required us to implement this simple form of register spill and unspill.

[6][Henry81] considers methods for finding alternative instructions.

5.4. Phase 4: Output Generation

Instructions are formatted into their assembly code representation by consulting two tables. The "print" field in the instruction table (figure 3) defines the operator. Each operand is printed by consulting a hand written *addressing mode format table*, which is not described here.

6. Grammar and Code Generator Design Details

We now examine issues in the grammar and code generator design, in particular, side effects, the grammar structure, the augmented grammar, our method for handling type conversion, and finally the problems we had "rounding out" the code generator to accept the entire "C" language. These issues relate to all phases of the code generator.

6.1. Side Effects

If an instruction produces useful results in two or more locations, then the basic Graham-Glanville algorithms can only track one result. The other result must be ignored, unless the computation has been flagged in the input to the code generator as a kind of common sub-expression. Failure to exploit such multi-valued expressions can result in code inefficiency for both auto-increment and auto-decrement side effects on registers (both are classed as the *autoinc* addressing mode), and for tracking the condition codes that are set by most instructions. We are able to model autoinc and condition codes by performing either source language driven recognition, or extremely simplified data flow analysis. For "C" programs on the VAX, we generate acceptable code.

We are currently examining the possibility of using a code generator that does not recognize autoinc or condition codes, together with a peephole optimizer with data flow analysis [Davidson81] [Giegerich82]. The peephole optimizer would introduce autoinc and condition code improvement where possible. That organization would simplify more parts of the code generator than the following discussion suggests.

The peephole optimizer strategy would still not model either the extended divide instruction, which computes both a quotient and a remainder, or the string instructions. However, using both quotient and remainder would be appropriate only if the source language provided a multi-valued divide operator, or if data flow analysis revealed that both values were used. Neither of these situations are possible in the compilers under discussion. [Morgan82] discusses string instructions in a companion paper.

In the present code generator, we generate the autoinc addressing mode only to modify dedicated registers, and then only if the dedicated register is the destination of a postfix increment or prefix decrement binary operator. These operators are in the intermediate language only if they are in the source language. Finding other opportunities for autoinc entails data flow analysis. [Ganapathi81] discovers ways to use autoinc by a post analysis of basic blocks.

The semantic descriptor for autoinc must be handled carefully to prevent the side effect from occurring more than once; after the first use and side effect, the descriptor is modified so any subsequent reference will refer to the same location[6].

On the VAX, almost every instruction sets the condition codes, usually as a side effect. Consequently, if a condition code value is to be used for a conditional branch (the only possible use), that use must follow immediately. Therefore, our code generator can handle condition codes syntactically without the multi-value tracking problem we alluded to earlier.

In fact, because of the immediacy of condition code use, condition codes are not explicitly reflected in our machine description. For example, an **add** instruction which puts its value into a register might have the description:

```
reg →    Plus rval rval    "addl3 rval, rval, reg"
```

where "rval" denotes an addressing mode. As a side effect, the instruction sets the condition code. A conditional branch instruction which tests the condition code is described by:

```
• →    Cbranch Cmp reg Zero Label    "jCmp Label"
```

Here terminals, "Cmp" and "Zero" describe the particular condition of interest and "reg" implicitly refers to the condition code setting. Thus, the pattern matcher automatically determines from context whether the instruction was executed for its condition code setting or not.

6.2. The Augmented Grammar

We developed the initial machine description grammar by factoring address subtrees and operator classes. This initial grammar was over factored, causing incorrectly resolved shift/reduce conflicts, leading to incorrect or inefficient code. The grammar also had syntactic blocks. Both problems were solved by adding productions to the initial grammar.

6.2.1. Overfactored Grammar

As an example of *overfactoring*, our initial factorization grouped the operators "Plus", "Mul", "Or", and "Xor" together into a special operator non-terminal, called "binop". This new non-terminal replaced the operator in occurrences in which it was the primary operation of the instruction. We chose this factorization to reduce the number of states. This factoring seemed to make sense, since these operators are all binary and commutative, have identical operand semantics, and are implemented in essentially the same way. However, "Plus" and "Mul" also occur in contexts in the grammar in which they are secondary operations, for example within addressing modes. Consequently, the initial grouping caused many shift/reduce conflicts of the (simplified) form:

```
[ displ → Plus • constant reg ]
[ binop → Plus • ]
```

A decision to shift in this state is tantamount to deciding that the "Plus" will be implemented by the addressing hardware as a displacement address ("displ"), rather than by an add instruction. The decision is premature, and could lead to a syntactic block, although not in this simple example. To avoid this problem, "Plus" and "Mul" cannot be factored as a "binop", although that factoring is valid for "Or" and "Xor".

The example describing condition codes in §6.1 is also over factored. There is a production, not previously shown,

```
reg →    Dreg    {no code, just condense}
```

that allows dedicated registers to be used everywhere assignable registers can be used. This production does not generate code, and so the condition codes are not set, contrary to the assumption about the non-terminal "reg" in the first general pattern:

```
• →    Cbranch Cmp reg Zero Label    "jCmp Label"
```

To solve this problem, we added a new production:

```
• →    Cbranch Cmp Dreg Zero Label
            "tstl Dreg; jCmp Label"
```

This change circumvents the factoring in this context, since the choice of a shift over a reduce will force selection of the second "Cbranch" pattern for a dedicated register, rather than the first. Notice that we still have the advantage of factoring in other register contexts.

Most of the outstanding bugs in our code generator are caused by remaining instances of overfactoring. We are developing a factoring theory to help us find and repair these cases automatically. We have not yet invested the time to implement tools and redesign the grammar to avoid overfactoring. However, we feel there is nothing inherently difficult about writing machine grammars to avoid these problems.

6.2.2. Syntactic Blocking

We also had to add productions to alleviate syntactic blocks (error productions). Syntactic blocks occur in long productions because coding alternatives, expressed as shared left context in a given state, are insufficient to handle all cases. To prevent syntactic blocking, we added *bridge productions* to the grammar. A bridge production shares left context to the point of, or slightly preceding, a syntactic block. A bridge production does not correspond to a single instruction or addressing mode. However, it will handle the more general continuation of the shared prefix.

[6]The descriptor may be reused once in a chained assignment operator of the form a = b = c; in this case the descriptor for b is used once as a destination, and once as a source.

The following two productions show how the bridge productions are used. The tree form of the right hand sides is in Figure 4; the linearized form is shown below.

```
block:   dx →  Plus Plus Const reg Mul • Four reg
bridge:  reg →  Plus Plus Const reg rval
```

The first production describes the address computation performed by the displacement indexed addressing mode; note that the computation performs an implicit multiplication by four. However, there is a syntactic block in this production at the left subtree of the "Mul", indicated by the "•", since other arguments to "Mul" are possible. The second production is the bridge production. "Rval" is a non-terminal corresponding to any general operand, and consequently any integer expression which could be computed into a register, so "rval" will *cover* the evaluation of subtrees other than multiplication by *four*.

Syntactic blocks can be detected automatically by the table constructor (§3.2), although when we developed the grammar, this part of the table constructor was disabled for trivial implementation reasons. We found it satisfactory to inspect the dense action-next tables for *error* actions; we had no surprises when we re-enabled the automatic check. The semantic actions for bridge productions can be synthesized automatically from real instructions, although we also did that by hand. (The automatic techniques are discussed in [Glanville78].)

We do not yet have a theory explaining what the unshared part of the bridge production should be. In the example above, we chose the recursive non-terminal "rval"; the recursive non-terminal "reg" would have covered the block as well. When the grammar is written it is difficult to predict which choice for the covering non-terminal produces better code.

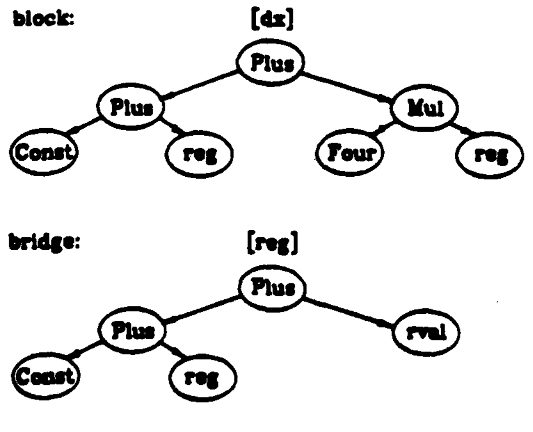


Figure 4: A Syntactic Block and its Bridge

6.3. Semantic Blocking

Our VAX description has no instances of semantic blocking. Only one situation arose in which semantic blocking might have occurred. Since the default list construction mechanism in CGGWS was disabled when we developed the code generator, we converted the semantic block to a syntactic block, which was then resolved by bridge productions. We describe this case, because it appears to represent a more general sort of choice between syntax and semantics.

In its simplest form, semantic blocking can arise in describing the addressing modes of the VAX because the hardware incorporates typed addressing. For example, the general form of the address computed by displacement indexing is:

```
dx →  Plus Plus Const reg Mul Const reg
```

where the "Const" left child to the "Mul" operator is semantically restricted to be 1, 2, 4 or 8 (corresponding to byte, word, long or double word addressing). If the second "Const" is not 1, 2, 4 or 8, a sequence of instructions must be generated. To avoid semantic blocking, we introduce the syntactic tokens "One", "Two", "Four", and "Eight" in place of the second "Const" when describing the addressing mode. The resulting syntactic block is resolved with a bridge production, as discussed in the previous section.

The replacement of the semantic constraint by a syntactic one potentially reduces the power of the code generator. Now, the special constants will be discovered only if they were converted to syntactic tokens when the input was generated. That might not be done for every occurrence of those constant values. On the other hand, it may be faster to handle these cases syntactically, rather than semantically. This issue merits further investigation.

6.4. Data Types and Type Conversion

The expression trees input to the code generator consist of generic operators attributed with the data type of the resulting value. Other attributes to the operators or operands, such as the binding of a dedicated register, need not concern us here. With the exception of the *de-referencing* operator and the incomplete set of conversion operators, the operands must be converted to have the same type as the expected result before the operation can be performed. This reflects the semantics of "C", Pascal and Fortran, as well as the conventions of the hardware. However, in the incoming expression trees, the operands have their own data types, which need not agree with the data type of the operator. The "front ends" rarely generate the conversion operators.

In this experiment we did not check any semantic attributes in the expression tree as a condition to selecting among productions in a reduce/reduce conflict, nor did we have the related mechanism for default list construction and application that the basic Graham-Glanville algorithm uses. Consequently, we had to use syntax to insure that the type of the actual operands correctly matched the expected type of the formal operands, and that appropriate conversion instructions were generated. This type matching is necessary before the instruction table (§6.3.1) is searched.

In order to implement type checking and type conversion syntactically, every symbol that can possibly have *n* different type attributes must be replaced by *n* different symbols, one for each type. In addition, the special constants 0, 1, 2, 4 and 8 must have their own terminal symbols, because of the importance they play in comparisons and address construction.

As a consequence of "syntax for semantics", we have separate instruction patterns for each machine type for which the operation is defined. Elaborating these new symbols and productions by hand is tedious and error prone. Instead, we write generic operators, operands and productions, and use a macro preprocessor to *type replicate* the generic grammar, yielding the final grammar from which the tables are constructed. The macros are three characters long. The first and third characters are identical, and tersely specify the set of machine data types permitting replication. The second character specifies which replication operator is to be applied.

For example, a generic production describing the address computed by displacement indexing, the example used in §6.2.2, is shown here[7]:

```
dx_#T# → Plus_l Plus_l Const reg_l Mul_l #V# reg_l R(7)
```

The character "#" indicates that the four machine types with different data sizes are valid; the class characters "T" and "V" expand to a type specific suffix character and constant, respectively. Type replication yields:

```
dx_b → Plus_l Plus_l Const reg_l Mul_l One reg_l     R(7)
dx_w → Plus_l Plus_l Const reg_l Mul_l Two reg_l     R(7)
 dx_l → Plus_l Plus_l Const reg_l Mul_l Four reg_l   R(7)
dx_d → Plus_l Plus_l Const reg_l Mul_l Eight reg_l   R(7)
```

Type replication has three drawbacks in our implementation. First, the size of the final grammar is enormous. Second, the type replicator only works on productions whose intra production type variation is consistent; it can not automatically expand data type cross products to create the sub-grammar describing the data conversion instructions. We performed this cross product by hand and introduced several errors.

The third problem with type replication, as we implemented it, is the interface between the grammar and the routines that encapsulate semantics. The interface is constrained by the function call to "R" that takes a single argument, a hand assigned production number. This restricted call to "R" is caused by bad design in the grammar specification language, and is not an inherent problem with our code generation technique.

It is clear that putting type constraints back into the semantic constraints would make the description smaller. What is less clear is the effect that would have on the speed of the code generator, since semantic checking is, in effect, interpretive. We intend to explore that time/space tradeoff.

6.6. Rough Edges

Most of the operators forming the intermediate expression trees were easily implemented on the VAX, as it was easy to describe most of the VAX instructions, operands and data types. For a number of reasons, we had difficulties implementing operators manipulating structures, fields, and unsigned numbers. The intermediate trees are both undocumented and poorly suited to expressing structures and fields. Some of these problems would go away if we introduced new operators in the trees, spent more time canonicalizing the tree before pattern matching, and redesigned some semantic routines.

The "C" language compound assignment operators, such as "+=", are expanded by the tree rewriter in the first phase. For example, the "C" statement

```
a += b
```

becomes:

```
a = a + b
```

Before "a" is treated as a common sub expression, it is forced to be directly addressable. This transformation introduces additional operators and requires mechanisms in the third phase to define and reference a locally scoped common sub expression corresponding to an address. This rewriting rule is compatible with our algorithm for finding binding idioms, as the assignment operator is first coded as a three address instruction, and then transformed into a two address instruction. Unfortunately, assignment operators not implemented by the hardware in one instruction, such as unsigned division, are not handled well by our model. Our only reason for performing this transformation was to minimize the number of patterns in the machine grammar, as we did not want to maintain separate patterns for variants of the same hardware instruction. In retrospect, the problems we created were much harder to solve correctly than those we avoided.

We spent inordinate amounts of time writing and testing expressions that exercise the union of problem areas in our code generator. As a bonus, we also found bugs in the portable "C" compiler. The complicated expressions we invented can not be expressed in Pascal or Fortran77. Our favorite was an expression with chained unsigned assignment division operators on auto incremented bit field operands!

7. Code Generator Development

We developed our code generator over approximately a year of part-time work. (Some of this time was spent improving the table generator.) R. R. Henry originally developed the code generator to check some hypotheses concerning grammar factorization. The machine description grammar was first written only for long data types and the easily implemented instructions, but included all addressing modes. When the factoring hypotheses were empirically verified, we began to extend the code generator so that it could generate code for interesting programs. It took us several

[7]Since "Plus" is commutative, and the patterns are [illegible] there are five other generic productions describing displacement indexing that we do not show.

iterations before we discovered the algorithms to transform and canonicalize trees.

R. Schulman re-wrote the grammar and semantic actions so instruction selection was data type sensitive. This addition was compounded by poorly understood inter-production interactions caused by overfactoring. We also changed the simple register manager to allocate double registers and to spill and unspill registers. Adding structure assignment and field operators affected all sub-phases in the code generator, most notably the instruction generator.

In the last stages of development, we became bogged down because it required over two memory-intensive hours of VAX 11/780 CPU time to construct a new set of tables from the enormous machine description grammar. Since we could only iterate on the grammar once per day, we removed bugs by modifying the grammar only as a last resort. Nevertheless, the table constructor was run more than 225 times during development, almost always for a data-type subsetted description grammar. Subsequently, we have developed new techniques which speed up the table constructor dramatically.

Because the code generator was developed by "augmentation", instead of by a grand plan, and the grammar developed by "avoidance", the internal semantics have many rough edges. We gave no thought to speed when coding the semantics, and find that the code generator runs slightly slower than the portable "C" compiler.

8. Code Generator Statistics

Our generic machine description grammar for the VAX, before type replication, has 458 productions, 115 terminals and 96 non-terminals. After type replication, the final grammar has 1073 productions, 219 terminals, and 148 non-terminals, and yields an instruction selector with 2216 states[8].

Our code generator spends most of its time parsing. This reflects both the large number of chain productions in the grammar, and the time spent manipulating and unpacking the description tables. Many of the chain productions are a consequence of the syntactic treatment of machine data-type conversion. The Appendix shows the actions the parser/pattern matcher makes when generating code for a simple Pascal statement.

Our code generator produces code that passes validation suites for "C", for Pascal and for Fortran77, although there are still subtle bugs involving conversion between signed and unsigned data types. For a particular large "C" program, our code generator generates code in 80.1 seconds, compared with the 56.4 seconds the portable "C" compiler spends[9]. Our coder produces 11365 lines of assembly code; PCC produces 11399 lines. Although we have not done any statistical comparisons, it appears that the code generated by our program is as good or better than that produced by the portable "C" compiler in almost all cases.

Naturally, our restricted code generation model affects the quality of code we produce. We could not change the "front ends" of the "C", Pascal or Fortran77 compilers, nor could we change the existing peephole optimizer or assembler that further process the assembly code the code generator produces. We do not perform data flow, or common sub-expression analysis, or any non-local code improvement. Finally, we omitted certain instructions from our machine description, such as the loop instructions.

9. Conclusions

We have not yet had any experience retargeting this compiler to other machines. We feel that the techniques to factor the machine grammar can be applied to a new machine. In a new implementation, we would reconsider our decision to type operands syntactically, a convention which greatly increases the size of the grammar.

We see two primary benefits of this experiment. First, in our experience, using a pattern matcher in a production compiler provides a well understood model for instruction matching. The pattern matcher is a convenient place to encapsulate, in a well understood way, almost all the knowledge about instruction patterns.

The experiment has also pointed up some of the important issues to pursue in developing this method further. We have already improved our algorithms for table construction so that the computation for our complete VAX description, which used to take over two hours, now takes ten minutes. We are investigating the tradeoffs between syntactic and semantic treatment of attributes. In that connection, we are studying the best way to use the formalized attribute processing proposed by [Ganapathi80]. We are also examining ways to recognize and handle situations in which maximal munch is, within our code generation model, suboptimal. We are examining the interaction between pattern-directed code generation with flow analysis and optimization, and the interface between our method for table-driven code generation and peephole optimization.

10. Acknowledgements

Stuart Feldman, Dan Halbert, Peter Kessler, Marshall K. McKusick, and Tom Morgan provided valuable comments on early drafts of this paper.

11. References

[Cattell80] Cattell, R.G., "Automatic Derivation of Code Generators from Machine Descriptions", *ACM Transactions on Programming Languages and Systems* 2(2), pp. 173-190 (April, 1980).

[Crawford82] Crawford, John. "Engineering a Production Code Generator", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, (June 23-25, 1982).

[Davidson81] Davidson, J.W. *Simplifying Code Generation Through Peephole Optimization*, PhD

[8]In comparison, a lambda free ADA grammar has 497 productions, 111 terminals, 304 non-terminals, with 908 states.

[9]The time for both compilers includes the time spent reading the intermediate file.

Dissertation, TR 81-19, Department of Computer Science, University of Arizona (December, 1981).

[Feldman79] Feldman, S.I., "Implementation of a Portable Fortran 77 Compiler Using Modern Tools", *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 14(8) pp 98-106 (August, 1979).

[Ganapathi80] Ganapathi, M. "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD Dissertation, TR #406, Computer Science Department, University of Wisconsin, Madison, WI (1980).

[Giegerich82] Giegerich, R. "Automatic Generation of Machine Specific Code Optimizers", in *Conf. Record Ninth ACM Symp. Principles of Programming Languages*, (January 25-27, 1982).

[Glanville77] Glanville, R.S. "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", PhD Dissertation, UCB CS-78-01, Computer Science Division, EECS, University of California, Berkeley (December, 1977).

[Glanville78] Glanville, R.S., and Graham, S.L. "A New Method for Compiler Code Generation", *Conf. Record Fifth ACM Symp. Principles of Programming Languages*, (January, 1978).

[Graham80] Graham, S.L. "Table Driven Code Generation", *IEEE Computer* 13(8), pp. 25-33 (August, 1980).

[Gujral81] Gujral, I. S. *Retargetable Code Generation for ADA*[10] *Compilers*, TP 127, Softech, Waltham, MA (December, 1981).

[Henry81] Henry, R.R. "The Code Generator Generator's Work Station: Experiments with the Graham-Glanville Machine Independent Algorithms for Code Generation", Master's Project Report, UCB/ERL M81/47, Electronics Research Laboratory, University of California, Berkeley (June, 1981).

[Jansohn80] Jansohn, H.S. and Landwehr R. "CGSS: Ein System zur Automatischen Erzeugung von Codegeneratoren", Universitat Karlsruhe, Karlsruhe, West Germany (July, 1980).

[Johnson79] Johnson, S.C. and Ritchie, D.M., "Portability of C programs and the UNIX System", *Bell System Technical Journal* 57(6), pp. 2021-2048 (July, 1979).

[Johnson81] Johnson, S.C. *Personal Communication*, (July, 1981).

[Joy79] Joy, W.N., Graham, S.L., Haley, C.B. *Berkeley Pascal User's Manual Version 1.1*, Computer Science Division, University of California, Berkeley. (April, 1979).

[Landwehr82] Landwehr, R., Jansohn, H.S., and Goos, G. "Experience with an Automatic Code Generator Generator", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, (June 23-25, 1982).

[Morgan82] Morgan, Thomas M. and Rowe, Lawrence A., "Analyzing Exotic Instructions for a Retargetable Code Generator", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, (June 23-25, 1982).

[Schulman82] Schulman, R.A. "A Reimplementation of the Second Pass of the Portable C Compiler", Master's Project Report, Electronics Research Laboratory, University of California, Berkeley (to appear).

[Sethi70] Sethi, R. and Ullman, J.D. "The Generation of Optimal Code for Expressions", *JACM* 17(4), pp. 715-728 (1970).

[10]ADA is a trademark of the U.S. Department of Defense.

12. Appendix: A Complete Code Generation Example

This appendix shows the code our code generator produces for the "example expression" in this incomplete Pascal program.

```
program appendix(output);
var a: integer;                { stored as a global name }
 ...
 procedure foo;
  var b: -128 .. 127;          { stored in the frame }
 begin
  ...
  a := 27 + b                  { example expression }
 end;
begin
 ...
 foo
end.
```

The first pass of the Berkeley Pascal compiler turns the example expression into an intermediate tree with this linearized prefix representation. The tree transformation phase does nothing with the tree, so it is passed to the pattern matcher "as is".

```
Assign_l                  long assignment
  Name_l: "a"             long global name
  Plus_l                  long addition
    Const_b: "27"         byte constant
    Indir_b               indirection to fetch a byte
      Plus_l              address (long) addition
        Const_b: "b"      byte constant
        Dreg_l: "fp"      long dedicated register
```

The code generator performs the following sequences of shift, reduce, and accept actions when generating code for the example expression.

action		on what	semantic action
shift		Assign_l	
shift		Name_l	
reduce	name_l →	Name_l	encapsulate
reduce	notype_lvali_l →	name_l	encapsulate
reduce	notype_lval_l →	notype_lvali_l	glue
reduce	lval_l →	notype_lval_l	give type
shift		lval_l	
shift		Plus_l	
shift		Const_b	
reduce	const_b →	Const_b	encapsulate
reduce	const_w →	const_b	glue
reduce	const_l →	const_w	glue
shift		const_l	
shift		Indir_b	
shift		Plus_l	
shift		Const_b	
reduce	const_b →	Const_b	encapsulate
reduce	const_w →	const_b	glue
reduce	const_l →	const_w	glue
shift		const_l	
shift		Dreg_l	
reduce	sreg_l →	Dreg_l	encapsulate
reduce	reg_l →	sreg_l	glue
reduce	disp →	Plus_l const_l reg_l	encapsulate operand
reduce	am_uncon_b →	disp	glue
reduce	am_b →	am_uncon_b	glue
reduce	notype_rval_b →	Indir_b am_b	encapsulate
reduce	rval_b →	notype_rval_b	give type
reduce	reg_l →	rval_b	emit "cvtbl b(fp),r0"
reduce	notype_rvali_l →	reg_l	encapsulate
reduce	notype_rval_l →	notype_rvali_l	glue
reduce	rval_l →	notype_rval_l	give type
reduce	asg_tree_l →	Assign_l lval_l Plus_l const_l rval_l	emit "addl3 r0,$27,a"
reduce	c_trees →	asg_tree_l	glue
reduce	tree →	c_trees	glue
reduce	• →	tree	glue
accept		•	glue